CPSC3120: Design Analysis of Algorithms Assignment 1

1. Order the following list of functions by their big-Oh notation. Simplify the function notation using basic rules for logarithms and exponents in order to make their relative complexity evident.

$6n\log n$ $O(n\log n)$	2^{100} $O(1)$	$\log\log n$ $\log\log n$	$\log^2 n$ $\log^2 n$	$2^{\log n}$ $O(n)$
2^{2n} $O(4^n)$	$5n$ $O(n)$	$n^{0.01}$ $O(\sqrt{n}\log n)$	$\frac{1}{n}$ $O(\log n)$	2^n $O(2^n)$
$3n^{0.5}$ $O(\sqrt{n})$	$4^{\log n}$ $O(n^2)$	$n^2\log n$ $O(n^2\log n)$	$\sqrt{\log n}$ $O(\log n)$	$n\log_4 n$ $O(n\log n)$
4^n $O(4^n)$				

Function	Simplified function	
$1/n$	$O(1)$	Slowest growing
2^{100}	$O(1)$	
$\log\log$	$O(\log\log n)$	
$\sqrt{\log n}$	$O(\log n)$	
$\log^2 n$	$O(\log^2 n)$	
$n^{0.01}$	$O(\sqrt{n})$	
$3n^{0.5}$	$O(\sqrt{n})$	
$2^{\log n}$	$O(n)$	
$5n$	$O(n)$	
$n\log_4 n$	$O(n\log n)$	
$6n\log n$	$O(n\log n)$	
$4^{\log n}$	$O(n^2)$	
$n^2\log n$	$O(n^2\log n)$	
2^n	$O(2^n)$	
2^{2n}	$O(4^n)$	Fastest growing
4^n	$O(4^n)$	

2. Suppose we perform a sequence of operations on a queue data structure. After every n operations we make a copy of the entire queue for debugging purposes. Show that the cost of n operations (including the copy) is O(n) using the accounting method.

n = amount of operations
K = size of queue
c = maximum cost of non-copy operation, constant

	real cost	Schema	Amortized cost
operation	c	$2 \cdot c$	$O(2c) \in O(1)$
copy	K	0	$O(K) \in O(1)$,
n operations + copy	$(n \cdot c) + K$	0	$O(2c)n + O(K) = O(2cn) + O(K) = O(n)$

No matter how many operations you do before the copy, total real cost can never exceed $n \cdot K \cdot c$, and as the two "increasing" variable parameter linear time operations take place seperately, its always

3. Suppose we have implemented a k-bit counter with a k-element binary array. The counter is initially 0. The only available operation is increment(A) which adds 1 to the current number.
    - What is the worst-case running time of increment?
    - What is the worst-case complexity for a sequence of k-increment?
    - Use the potential method to find a better estimate.

- Worst case running time is $\Theta(K)$ – one operation per bit (need to flip all)
- Worst case complexity for sequence of increments is $O(kn)$
    - One operation per bit, for each increment

n = increment_num

ϕ = flips required

$$\phi(n) = \sum_{i=0}^{k-1} (n/2^i)$$

m = total increments

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$

$$= 0 + \phi(n) - \phi(n-1)$$

$$= 0 + \left(\sum_{i=0}^{k-1}\left(\frac{n}{2^i}\right)\right) - \left(\sum_{i=0}^{k-1}\left(\frac{n-1}{2^i}\right)\right)$$

Cost of increment(A) $= \sum_{i=0}^{k-1} \frac{n}{2^i}$, n = # being incremented to, k = bits

Cost of increment_seq(A, m)

$$= \sum_{n=0}^{m} \sum_{i=0}^{k-1} \frac{n}{2^i}$$

n = # being incremented to / m = total increments requested

```cpp
#include <iostream>
#include <cmath>

int calc_flips(int current_increment, int k) {
    int pred_flips = 0;
    for(int i=0; i<k-1; i++) {
        pred_flips = pred_flips + (current_increment / std::pow(2, i));
    }
    return pred_flips;
}

void Increment(bool a[], int k, int &current_increment, int &total_flips) {
    current_increment++;
    int flips = 0;
    for(int j=0; j<k; j++) {
        std::cout << a[j];
    }
    std::cout << "\n";
    std::cout << "Predicting " << calc_flips(current_increment, k) << " flips on this increment\n";
    for(int i=k-1; i>=0; i--) {
        if(a[i] == 0) {
            std::cout << "  Detected and flipped 0 to 1 at a[" << i << "]\n";
            a[i] = 1;
            flips++;
            total_flips++;
            break;
        } else {
            std::cout << "  Detected and flipped 1 to 0 at a[" << i << "]\n";
            a[i] = 0;
            flips++;
            total_flips++;
        }
    }
    std::cout << "      On increment number " << current_increment << ", flipped " << flips << " times to make " << total_flips << " total.";
    std::cout << "  a[0] currently " << a[0] << " \n";
}

int main(){
    int k = 8;
    bool binary_array[k] = {0,0,0,0,0,0,0,0};
    int n = 240;
    int current_increment = 0;
    int total_flips = 0;
    for(int i=0; i<n; i++) {
        Increment(binary_array, k, current_increment, total_flips);
    }
    for(int j=0; j<n; ++j) {
        std::cout << binary_array[j];
    }
}
```

4. Suppose we have 20 singleton sets, numbered 0 through 19, and we call the operation union(find(i),fin(i+5)), for i = 0,1,2,…..,14.
Draw a picture of the tree-based representation of the sets that result, assuming we don't implement the union-by-size and path compression techniques.

i = 0 => Union(Find(0), Find(5)) => 0 ← 5

i = 1 => Union(Find(1), Find(6)) => 1 ← 6

i = 2 => Union(Find(2), Find(7)) => 2 ← 7

i = 3 => Union(Find(3), Find(8)) => 3 ← 8

i = 4 => Union(Find(4), Find(9)) => 4 ← 9

i = 5 => Union(Find(5), Find(10)) => 5 ← 10

i = 6 => Union(Find(6), Find(11)) => 6 ← 11

i = 7 => Union(Find(7), Find(12)) => 7 ← 12

i = 8 => Union(Find(8), Find(13)) => 8 ← 13

i = 9 => Union(Find(9), Find(14)) => 9 ← 14

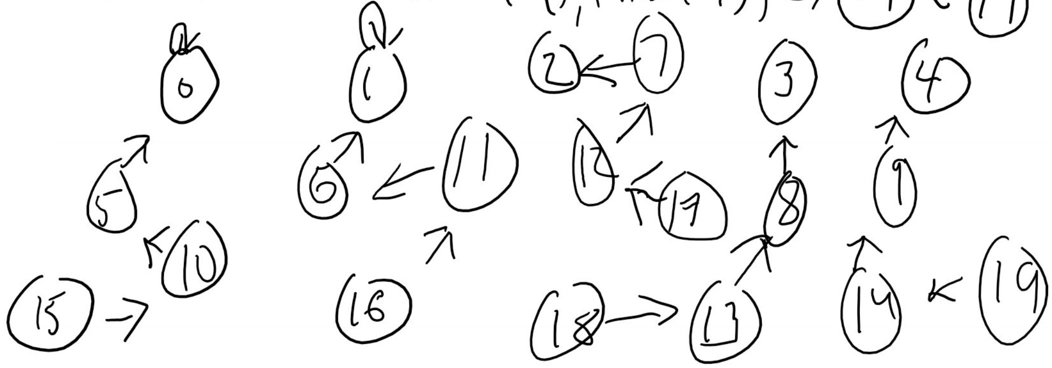
i = 10 => Union(Find(10), Find(15)) => 10 ← 15

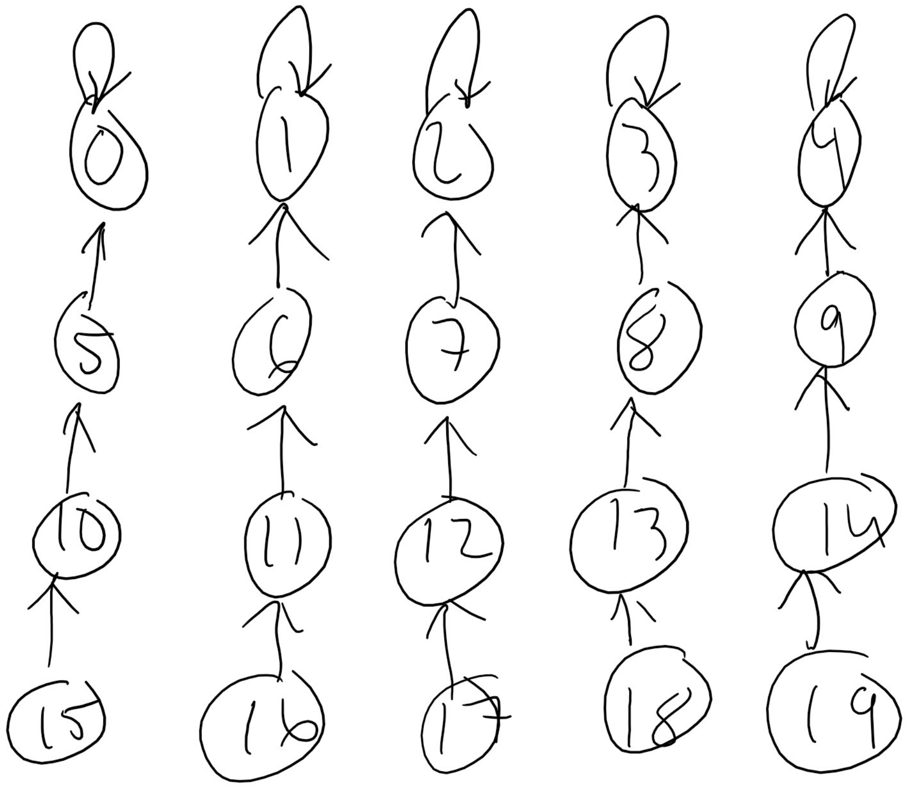
i = 11 => Union(Find(11), Find(16)) => 11 ← 16

i = 12 => Union(Find(12), Find(17)) => 12 ← 17

i = 13 => Union(Find(13), Find(18)) => 13 ← 18

i = 14 => Union(Find(14), Find(19)) => 14 ← 19





Answer for 4

5. Repeat exercise (4) assuming that we now implement both techniques